Army Research Laboratory

Computer Model for Manipulation of a Multibody System Using MATLAB

by Benjamin J. Flanders

ARL-TR-6594 September 2013

Army Research Laboratory
Aberdeen Proving Ground, MD 21005-5066

ARL-TR-6594 **September 2013**

Computer Model for Manipulation of a Multibody System Using MATLAB

Benjamin J. Flanders
Weapons and Materials Research Directorate, ARL

REPORT DOCUMENTATION PAGE						*Form Approved* *OMB No. 0704-0188*
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing the burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. **PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.**						
1. REPORT DATE *(DD-MM-YYYY)* September 2013	**2. REPORT TYPE** Final					**3. DATES COVERED (From - To)** March 2013
4. TITLE AND SUBTITLE Computer Model for Manipulation of a Multibody System Using MATLAB						**5a. CONTRACT NUMBER**
						5b. GRANT NUMBER
						5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S) Benjamin J. Flanders						**5d. PROJECT NUMBER** AH80
						5e. TASK NUMBER
						5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) U.S. Army Research Laboratory ATTN: RDRL-WML-A Aberdeen Proving Ground, MD 21005-5066						**8. PERFORMING ORGANIZATION REPORT NUMBER** ARL-TR-6594
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)						**10. SPONSOR/MONITOR'S ACRONYM(S)**
						11. SPONSOR/MONITOR'S REPORT NUMBER(S)
12. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution is unlimited.						
13. SUPPLEMENTARY NOTES						
14. ABSTRACT This report discusses a program that was written in order to support the FragFly model. The program assists users of the FragFly model in visualizing and posing a human target composed of moveable parts. The report discusses the architectural design of the posing program and identifies the specific format for constructing input files. The report also provides a user's guide for a graphical user interface that was created.						
15. SUBJECT TERMS body, simulation, multibody, pose, vertices, faces						
16. SECURITY CLASSIFICATION OF:			**17. LIMITATION OF ABSTRACT**	**18. NUMBER OF PAGES**		**19a. NAME OF RESPONSIBLE PERSON** Benjamin J. Flanders
a. REPORT Unclassified	**b. ABSTRACT** Unclassified	**c. THIS PAGE** Unclassified	UU	34		**19b. TELEPHONE NUMBER** (*Include area code*) 410-278-4257

Standard Form 298 (Rev. 8/98)
Prescribed by ANSI Std. Z39.18

Contents

List of Figures

List of Tables

1. Background

This report discusses a program that was written in order to support the FragFly model. The FragFly model was created to assess incapacitation of human targets given a detonation of a fragmenting munition. The model creates fragments using a ZDATA file (see "Proposed Model for Fragmentation using ZDATA Files");[1] the fragments are flown out to the surface of the target, and the model Operational Requirements-based Casualty Assessment (ORCA) is used to determine if an incapacitation has occurred. This is done by supplying the ORCA program interface an entry and exit location of each fragment through the body parts. ORCA assumes that the body and all its parts have a particular location and orientation. However, one of the objectives of the FragFly model was to allow the target to be posed into any posture. As a compromise, rigid-body rotations of body parts about joints are allowed to achieve intended postures. Rigid-body rotations by definition have inverse rotations. Therefore, regardless of the posture of the body, entry, and exit points of a fragment through the parts of a postured body can be translated back into the ORCA reference frame. This overcomes the potential limitations of using ORCA to determine incapacitation.

To do this a *multibody system*[2] was needed to properly represent the body with moveable parts such as arms and legs that can be rotated around joints like shoulders and knees. A multibody system is defined as an assembly of two or more rigid bodies imperfectly joined together, having the possibility of relative movement between them. Such a system was created; however, it lacks any kinematic behavior traditionally associated with multibody systems. A graphical user interface (GUI) tool was created to provide users a visualization of a body and provide real-time feedback for posing the body parts.

This report is broken down into five main sections.

1. Section 2. *Introduction to the Multibody System*.
2. Section 3. *Input Files.* This section introduces the readers to the definitions of the system and how the system is represented in data files.
3. Section 4. *GUI User Guide*. This section explains how to use the GUI created to pose the body.
4. Section 5. *Fundamentals of Rotations, Posing, and Postures.* This section explains the technical and mathematical details of the GUI and informs the reader how the FragFly model uses the output files of the GUI.

[1] Flanders, B. J. *Proposed Model for Finding Initial Conditions of Fragments from a Detonated Munition Using a ZDATA File*; U.S. Army Research Laboratory: Aberdeen Proving Ground, MD, to be published, 2013.

[2]http://mat21.etsii.upm.es/mbs/bookPDFs/Chapter01.pdf (accessed May, 2013).

5. Section 6. *Conclusions*. At this section the reader should be able to:
   - Recognize and create the GUI input files.
   - Launch (or start) the GUI.
   - Know how to use the interface in order to manipulate the moveable parts of the target.
   - Recognize and understand the output files of the posing program and how they will be used in the FragFly model.
   - Understand the mathematical expression used to perform the manipulations.

2. Introduction to the Multibody System

There is quite an extensive body of work related to multibody systems, which has application in the gaming industry, computer simulations, and robotics to name a few domains. A rather simple system was created to capture core functionality that is needed for the FragFly model. There are four major tasks of the system (which are implemented in the GUI).

1. Read input files.
2. Visualize the body.
3. Manipulate the moveable parts of the body (called posing) with real-time feedback.
4. Allow user to “commit” a posture by writing the output files.

These tasks can be seen in figure 1 as they are implemented in the GUI.

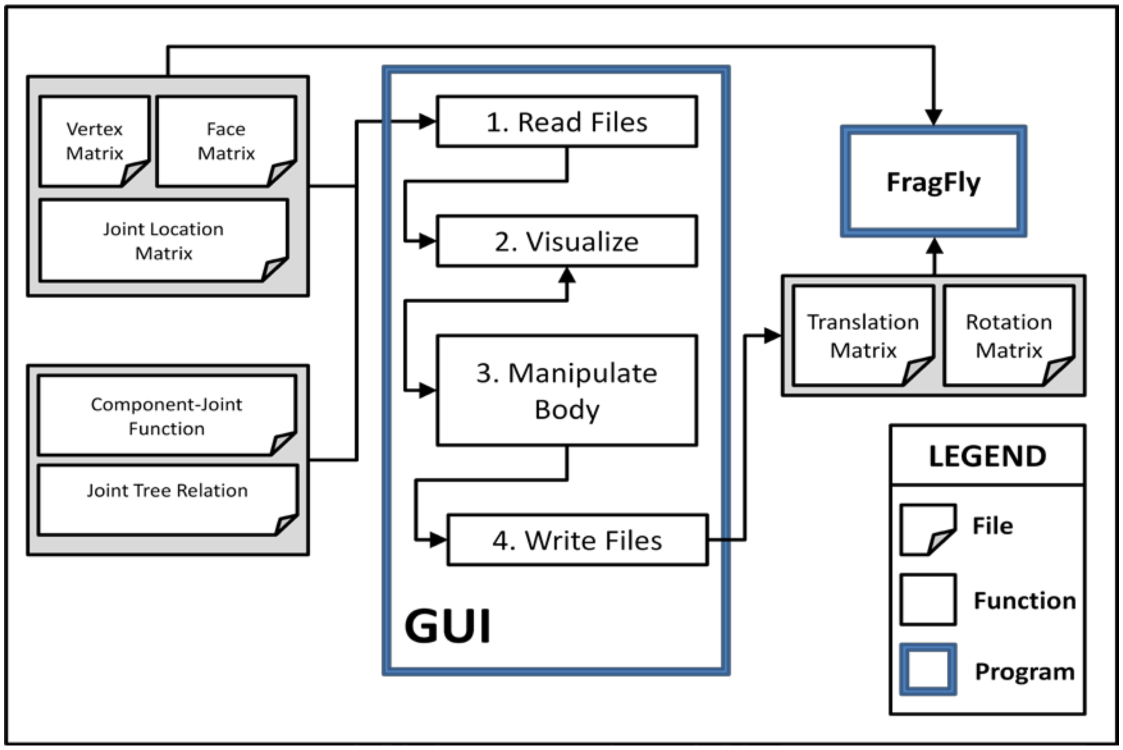

Figure 1. Multibody system framework.

The input files are shown on the left. These files are read in by the GUI program and stored into local memory. It also creates two more data structures called the Translation and Rotation matrices. They begin within local memory and are the only pieces of data that are changed throughout the lifetime of the GUI. The GUI provides the user a visualization of the body ("Visualize") and user interface buttons to rotate the parts of the body about the joints ("Manipulate Body"). Notice the double arrow between the functions. This means that as the body is being manipulated, the visualization of the body is updated along with the Translation and Rotation matrices. The process ends with the user electing to "commit" a posture (i.e., the visual orientation of the parts of the body), which causes the GUI program to write out the Translation and Rotation matrices. These output files, along with the Vertex, Face, and Joint Location matrices become inputs for the FragFly model. Note that at no point does the GUI program change the Vertex, Face, or Joint Location matrices.

3. Input Files

Collectively the input files, and the data they contain, represent the user's full understanding of the body within the system. It is expected that many users will not have all the files needed prior to reading this report. However, by the end of this section, the reader should be able to recognize how to create any missing input files.

The data in the system can be categorized into two types: declarative and relational. Declarative data represents values that can be measured or quantified. One example of this is a location or “point” in Euclidean space. Relational data represent mathematical functions or relations, which cannot be replicated using standard mathematical functions.

Below is a complete list of the input files separated by their type.

1. Declarative
   a. Vertex Matrix
   b. Joint Location Matrix
   c. Translation Matrix
   d. Rotation Matrix
2. Relational
   a. Face Matrix
   b. Joint Tree Relation
   c. Component-joint function

Each of the files will now be discussed.

3.1 Declarative Data

Following is a list of the input files that contain nonrelational data. A brief description of the file is given, along with suggested file names in parentheses. These names will also be used later when displaying code snippets. In general, files should not contain any headers or text (except for delimiters), are required to be comma separated, and use “newline” characters to separate rows of information.

1. Vertex Matrix (“vertices.txt”)—lists the points that lay on the surface of each subcomponent. It is assumed that all the vertices of the body are stored in one file, called the vertex file. For a body comprised of a total of *n* vertices, the file will have *n* lines and have (at least) four comma-separated columns. Additional columns will be ignored. The first three columns of a row are the *x*, *y*, *z* coordinates of a vertex, respectively, and the fourth column is the identification number (ID) of the component (a more general term for body part) the vertex belongs to. Figure 2 shows a snippet of a vertices file with the columns labeled.

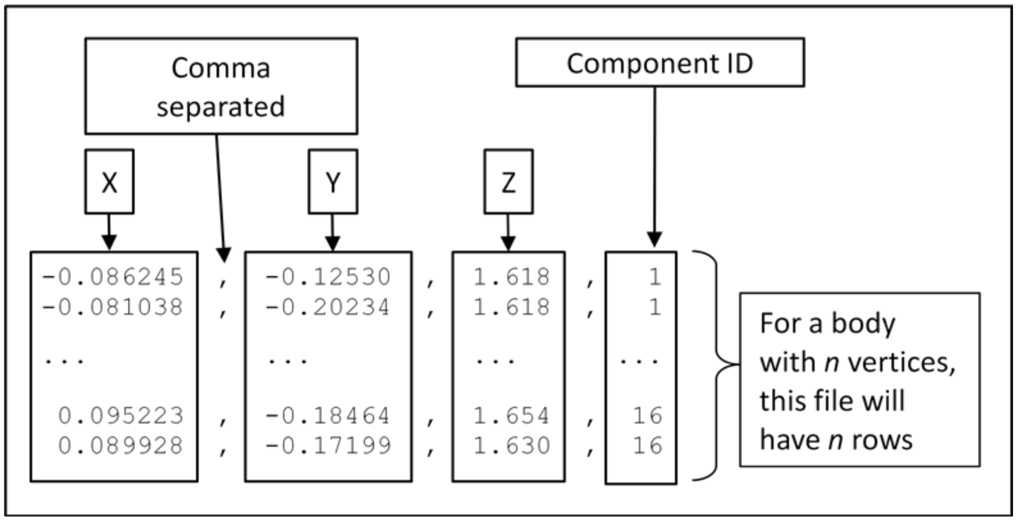

Figure 2. File description for the Vertex Matrix file ("verts.txt").

2. Joint Locations Matrix ("joints.txt")—lists the location of each joint. For a body with *j* joints, the joint location file will have *j* rows and three columns. The three columns are the *x*, *y*, *z* coordinates of each joint, respectively. Figure 3 shows a snippet of a joint location file with the columns labeled. The order that the joints are listed is important because it imposes on each joint an ID number. The ID numbers will be referenced in the relational data.

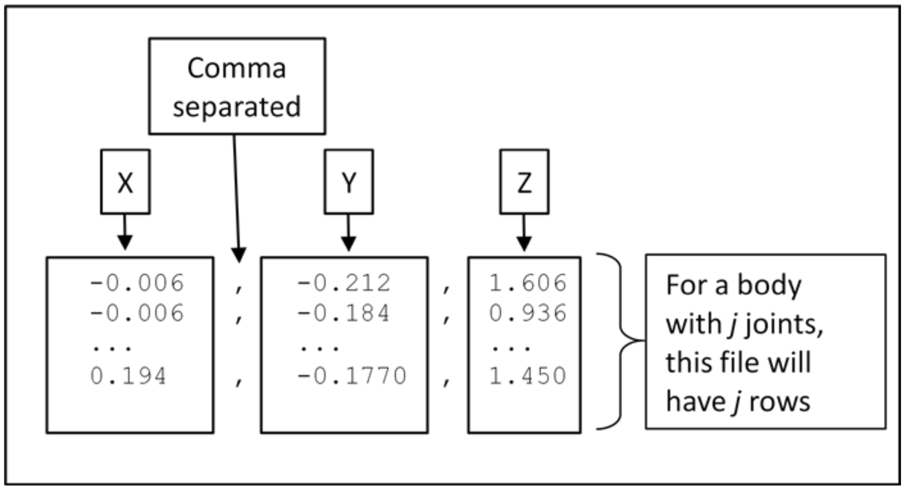

Figure 3. File description for the Joint Location Matrix ("joint.txt").

3. Translation Matrix ("translations.txt")—lists all the global translation for each joint. Essentially, each joint is assigned a translation vector, which initially has all zero entries. As the joint is relocated, the translation vector for the joint is updated to reflect the change in location from the original location. All the translation vectors are collected into one matrix. For a body with *j* joints, the translation matrix will have *j* rows. The *i*th row of the translation matrix will correspond to the *i*th joint in the Joint Location Matrix. The

Translation Matrix is first created in the local memory of the GUI program and is written out when the user "commits" a posture. Again, for a body with *j* joints, the file will consist of *j* rows and three columns. The three columns represent the *x*, *y*, *z* components of each translation, respectively. Figure 4 shows a snippet of the Translation Matrix as it would appear if it were written out at initialization.

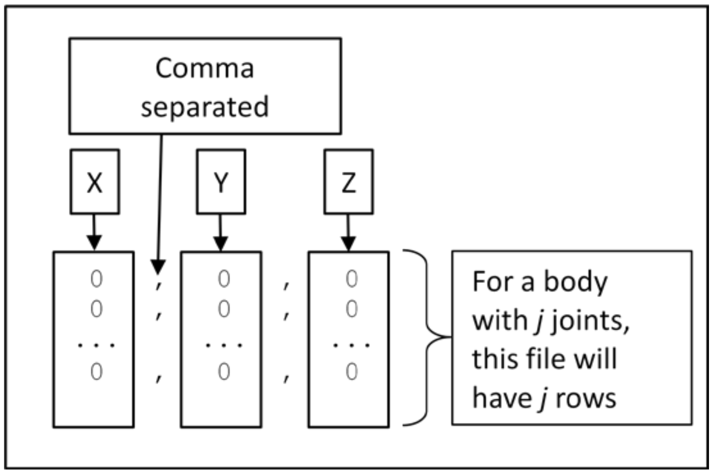

Figure 4. File description of the Translation Matrix ("translation.txt").

4. Rotation Matrix ("rotations.txt")—lists the coordinate system matrix for each joint. This matrix is first created in the local memory of the GUI program and is written out when the user "commits" a posture. Initially each joint is assigned the three-dimensional identity matrix as its rotation matrix. In order to store all the matrices for all the joints together, each matrix is "vectorized." Vectorizing is a process of reshaping a two-dimensional matrix into one dimension. A vectorized matrix is one that lists all the elements of the matrix along one row such as:

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \rightarrow [1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9]. \tag{1}$$

Figure 5 shows a snippet of what the Rotation Matrix file would look like if it were written out during initialization.

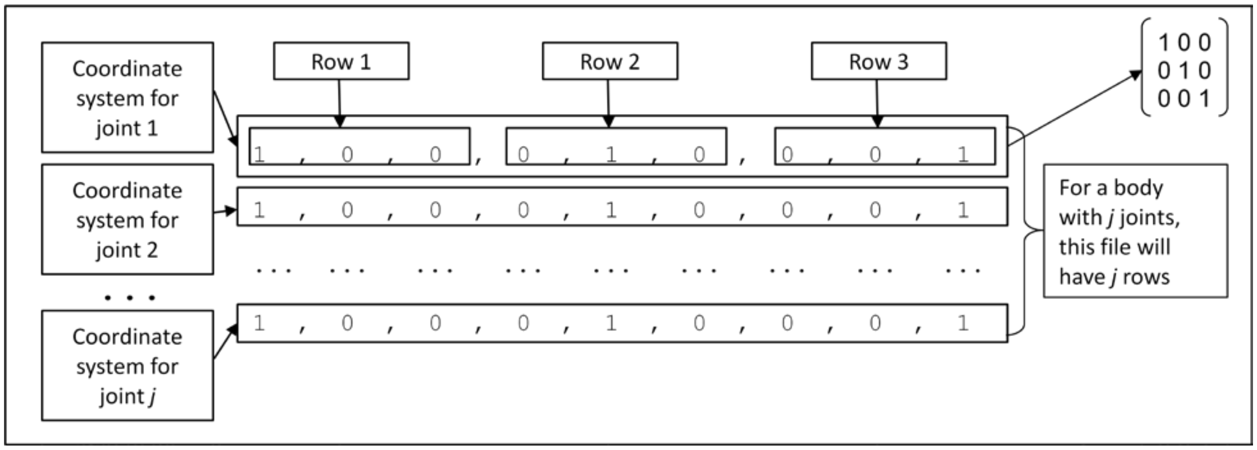
Coordinate system for joint 1
Coordinate system for joint 2
• • •
Coordinate system for joint *j*
Row 1
Row 2
Row 3
1 , 0 , 0 , 0 , 1 , 0 , 0 , 0 , 1
1 0 0
0 1 0
0 0 1
For a body with *j* joints, this file will have *j* rows

Figure 5. File description of the Rotation Matrix ("rotations.txt").

To provide a visualization of some of the declarative data, figure 6 shows the human body used in the FragFly model (see "An Alternative Representation of a Simulated Human Body").[3] Each component and joint is labeled and table 1 provides a description for each.

[3] Flanders, B. J. *An Alternative Representation of a Simulated Human Body*; U.S. Army Research Laboratory: Aberdeen Proving Ground, MD, to be published, 2013.

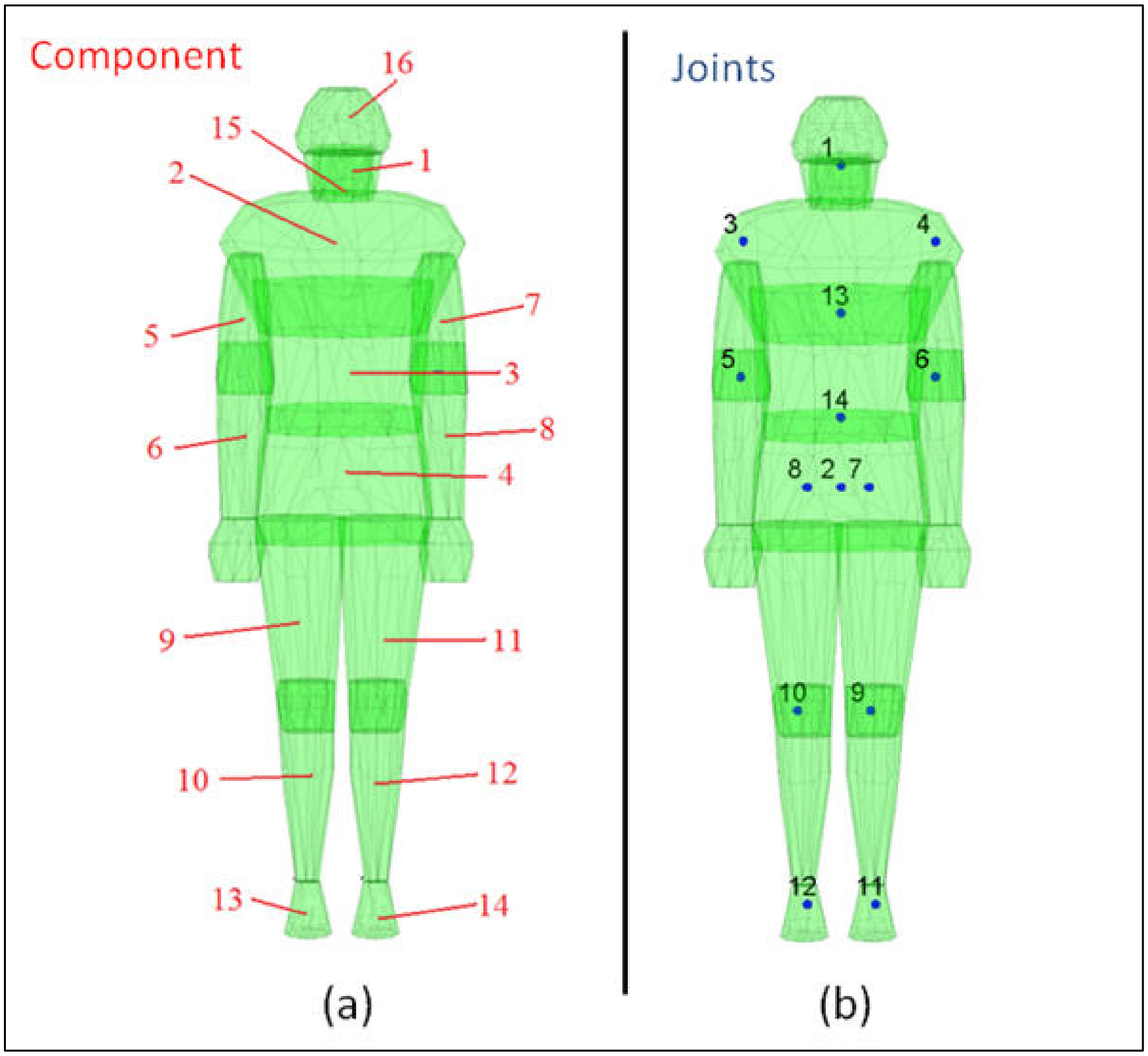

Figure 6. Human body example. Subcomponents and joints are labeled.

Table 1. Description of subcomponents and joints by ID.

Component		Joint	
ID	**Description**	**ID**	**Description**
1	Lower head	1	Neck
2	Thorax	2	Pelvic
3	Abdomen	3	Right shoulder
4	Pelvis	4	Left shoulder
5	Upper right arm	5	Right elbow
6	Lower right arm	6	Left elbow
7	Upper left arm	7	Left hip
8	Lower left arm	8	Right hip
9	Upper right leg	9	Left knee
10	Lower right leg	10	Right knee
11	Upper left leg	11	Left ankle
12	Lower left leg	12	Right ankle
13	Right foot	13	Thoracic vertebra 2
14	Left foot	14	Thoracic vertebra 10
15	Neck	—	—
16	Upper head	—	—

3.2 Relational Data

Following is a list of the input files that contain relational data.

1. Face Matrix (“faces.txt”)—used to connect vertices together to form surfaces of the target. For a body with a total of *k* faces, the face file has *k* rows and (at least) four columns. The first three columns of a row in the face matrix identify the three vertices that form a triangle surface by their row number in the Vertex Matrix. The fourth column represents the component identification number that denotes assignment. For instance, if a row in the face file reads [1, 17, 28, 5], then this face is formed by connecting the first, seventeenth, and twenty-eighth vertices found in the Vertex Matrix into one triangle. This face belongs to the component with the ID of 5. Figure 7 shows a template of the face file.

2. Joint Tree Relation (“joint_tree.txt”)—this file represents the relationship that the joints share with each other. This relationship is best described as an ordered tree relationship. Note that a body may contain any number of root nodes as illustrated in figure 6, which shows the joint tree for a body model in figure 8.

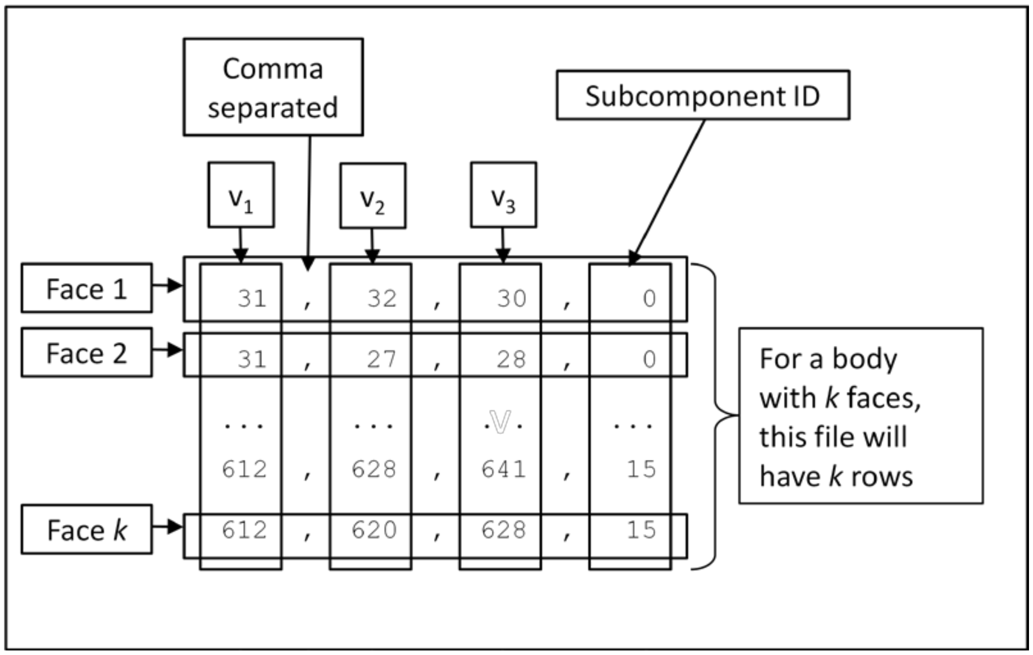

Figure 7. File description of Face Matrix ("faces.txt").

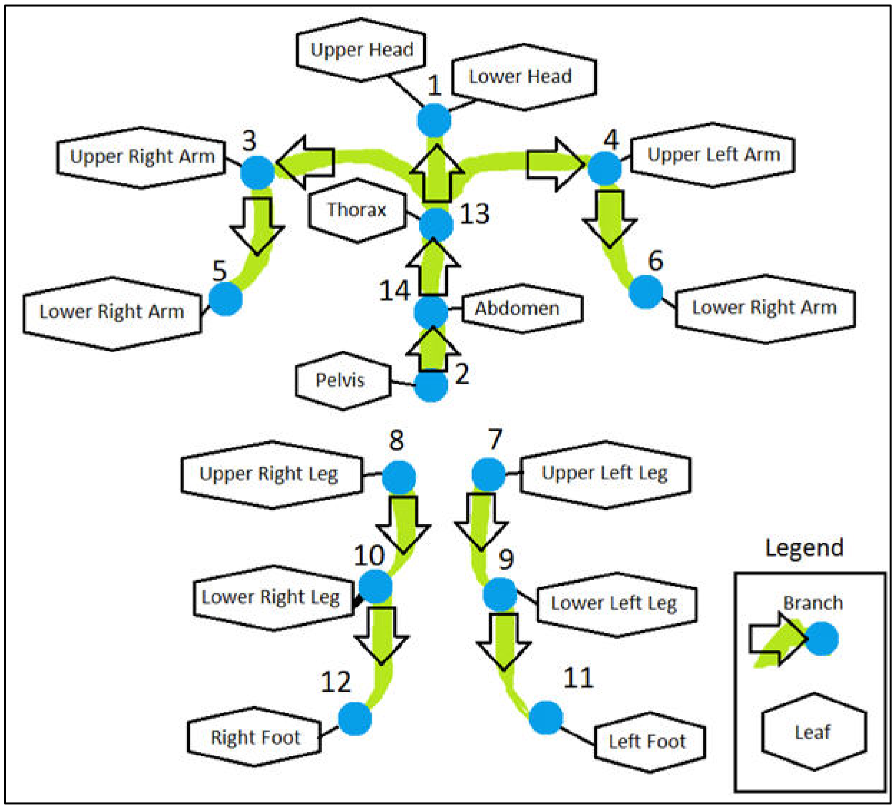

Figure 8. Joint tree with component leaves.

There are many ways to represent this structure conceptually. One is to define a relation so that

$$J_{tree}(i,j) = \begin{cases} 1 & \text{if joint } j \text{ is a child of joint } i \\ 0 & \text{otherwise} \end{cases}. \tag{2}$$

To be more explicit, one can say that $J_{tree}(i,j)$ is true if and only if the joint that is located in the j^{th} row in the "joints.txt" matrix is a child of the joint that is located in the i^{th} row of the same matrix. This relation can be expressed by forming, for a body with *j* joints, a square matrix that has *j* rows and columns. The entry of the i^{th} row and j^{th} column will be the value of $J_{tree}(i,j)$. Table 2 shows the entire file. Just like all other files, the values are comma separated and rows are separated by a newline character.

Table 2. Joint Tree Relation Matrix.

		Joint													
		1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**13**	**14**
Joint	**1**	1	0	0	0	0	0	0	0	0	0	0	0	0	0
	2	1	1	1	1	1	1	0	0	0	0	0	0	1	1
	3	0	0	1	0	1	0	0	0	0	0	0	0	0	0
	4	0	0	0	1	0	1	0	0	0	0	0	0	0	0
	5	0	0	0	0	1	0	0	0	0	0	0	0	0	0
	6	0	0	0	0	0	1	0	0	0	0	0	0	0	0
	7	0	0	0	0	0	0	1	0	1	0	1	0	0	0
	8	0	0	0	0	0	0	0	1	0	1	0	1	0	0
	9	0	0	0	0	0	0	0	0	1	0	1	0	0	0
	10	0	0	0	0	0	0	0	0	0	1	0	1	0	0
	11	0	0	0	0	0	0	0	0	0	0	1	0	0	0
	12	0	0	0	0	0	0	0	0	0	0	0	1	0	0
	13	1	0	1	1	1	1	0	0	0	0	0	0	1	0
	14	1	0	1	1	1	1	0	0	0	0	0	0	1	1

3. Component-joint function ("comp_function.txt")—notice that in figure 8 the components described in table 1 appear attached to a joint. Continuing the tree analogy, the subcomponents can be thought of as the leaves on the joint tree. Leaves can only belong to one branch, and in the same way, a subcomponent can only be attached to one joint. Thus one can define a function $C_{joint}(c)$ that maps the set of subcomponents $Components \rightarrow Joints$ so that

$$C_{joint}(c) = j \tag{3}$$

if and only if the subcomponent s is attached to joint *j*. It is left as an exercise to the reader to form the component joint function for the tree in figure 8.

4. GUI User Guide

Once the user has correctly created all the input files, he is then ready to use the GUI. The GUI program was written in MATLAB.* The language was chosen because of its three dimensionally graphing and viewing capabilities. The GUI program is a collection of data files and MATLAB scripts. All the files and scripts should be collocated in one file directory for convenience (this avoids the user from being required to specify file paths). The MATLAB current directory should be set to the file directory containing all the files.

There are three actions that a user can perform:

1. START: this will launch the GUI.
2. POSE: control mechanisms allow the user to pose the object.
3. COMMIT: after posing is complete, the user can write out the posture files.

Each action will now be discussed in detail.

4.1 START

Assuming that the files in the previous section are named accordingly, then in the "Command Window" type:

```
>> pose_body('verts.txt', 'faces.txt', 'joints.txt', 'joint_tree.txt',
'comp_function.txt')
```

The order of files is important. The file names should be in the following order: Vertex Matrix, Face Matrix, Joint Location Matrix, Joint Tree Relation, and Component-Joint Function.

This will execute the function called "pose_body," which will produce the window as seen in figure 9.

The user may prefer to maximize the window, which will make the layout of the window less cluttered.

* MATLAB is a registered trademark of The MathWorks, Inc.

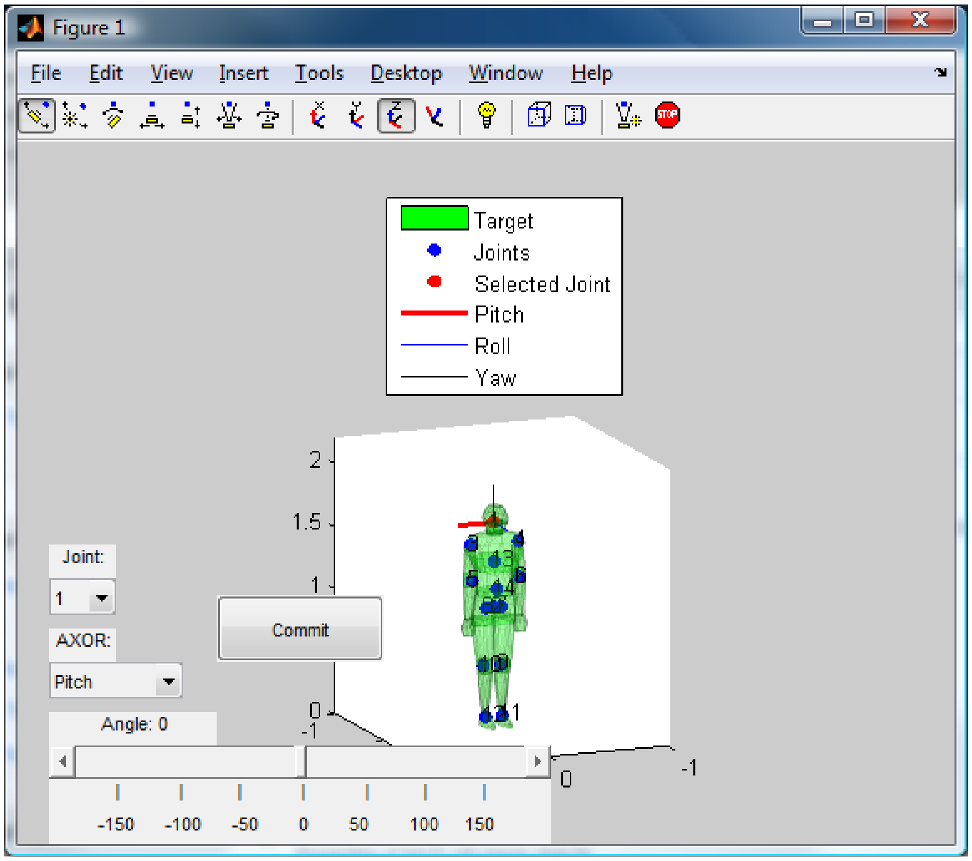

Figure 9. Startup window for GUI.

4.2 POSE

To pose the body, follow the steps in figure 10.

1. Select a joint from the drop-down menu labeled “Joint.” The selected joint will appear as a red dot. The axis of rotation will be redrawn on the joint selected and all components that will be affected by the joint rotation appear as more solid than the unaffected components.
2. Select an axis of rotation from the drop-down menu labeled “AXOR.” The selected axis will appear in bold. The three axes of rotation for a joint will rotate along with the joint. The axis of rotation is actually taken from the coordinate system matrices.
3. Modify the angle of rotation by using the slider labeled “Angle.”

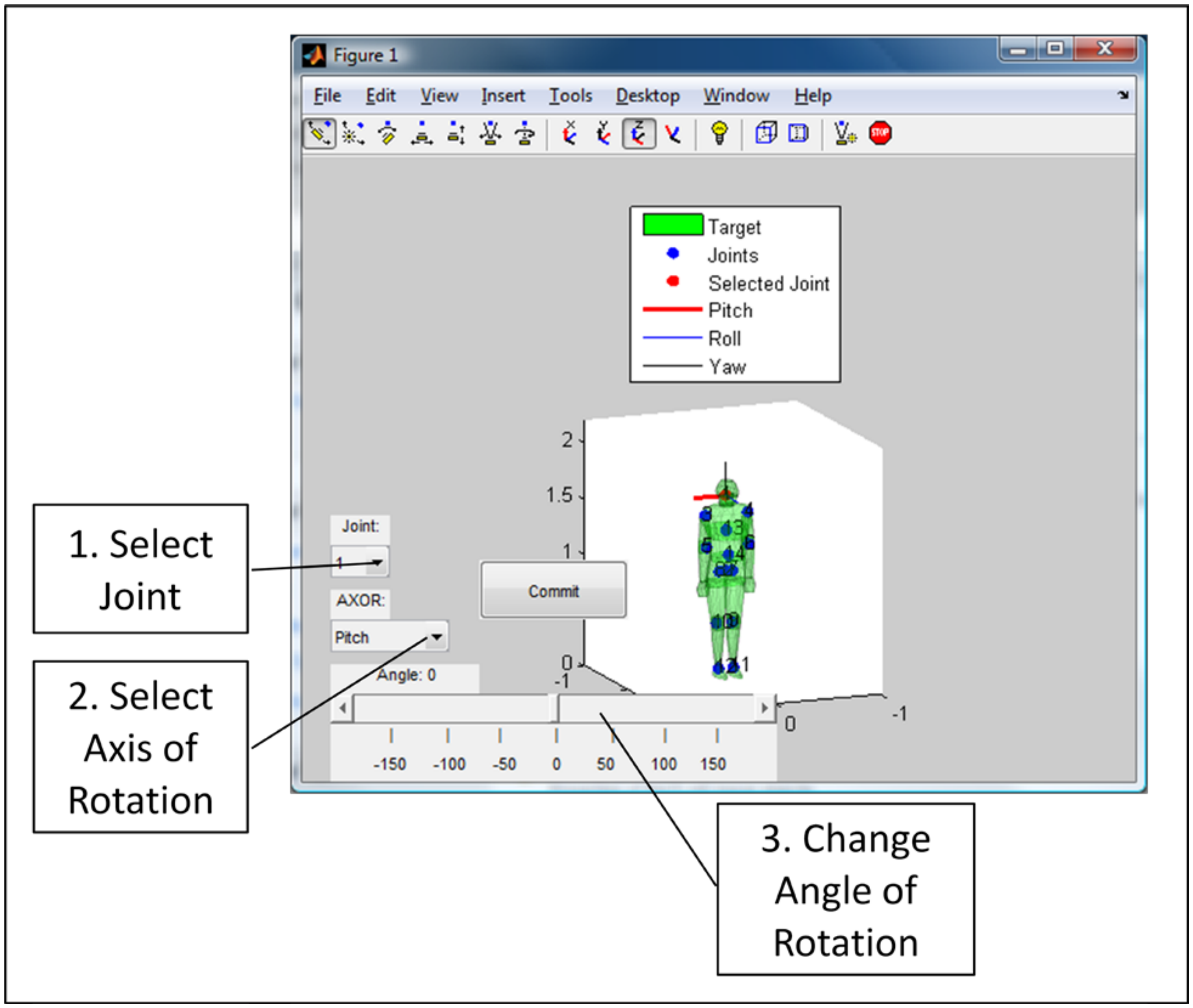

Figure 10. Steps to pose an object.

By default the MATLAB camera toolbar is opened. This allows the user to manipulate the view of the target to a desired point of view. Camera actions do not pose the object or affect the posture.

4.3 COMMIT

After the body is in a desired posture, the user can "commit" the posture by pressing the "Commit" button. As mentioned before, pressing this button will write out the Translation and Rotation matrices into files. Before the files are written out an adjustment is made to the translation vectors. The adjustment is to ensure that the lowest point on the body (the vertex with the least *z* component value) is incident with the *x*-*y* plane. The user will be prompted to provide file names for each posture file. Default values are given. These files are ready to be used in the FragFly model.

5. Fundamentals of Rotations, Posing, and Postures

Rotating a body part about a joint (called a joint rotation) is a multistep process. In a joint rotation one must be able determine which components *and* joints will be affected. For instance a shoulder rotation will rotate the entire arm as well as the elbow joint. If a joint is rotated, its translation vector and rotation matrix must be updated in the Translation and Rotation matrices, respectively. Before the joint rotation algorithm is discussed, three key concepts will be presented:

1. Rotate a point about the origin
2. Rotating a point about another point given a rotation matrix
3. Rotate a joint's rotation matrix.

These concepts will help support three main algorithms:

4. Posing Algorithm
5. Posture Algorithm
6. Reverse-Posture Algorithm.

5.1 Rotate a Point About the Origin

The simplest rotation is one about the origin. Rotations about the origin are defined by an axis of rotation (required to be a unit vector), call it $\vec{u}$, and an angle of rotation, call it θ. The vector $\vec{u}$ and angle θ can be used to form a matrix *R*,

$$R = f(u,\theta) = \begin{bmatrix} \cos\theta + u_x^2 \cdot d & u_x \cdot u_y \cdot d - u_z \cdot \sin\theta & u_x \cdot u_z \cdot d + u_y \cdot \sin\theta \\ u_x \cdot u_y \cdot d + u_z \cdot \sin\theta & \cos\theta + u_y^2 \cdot d & u_y \cdot u_z \cdot d - u_x \cdot \sin\theta \\ u_x \cdot u_z \cdot d - u_y \cdot \sin\theta & u_y \cdot u_z \cdot d + u_x \cdot \sin\theta & \cos\theta + u_z^2 \cdot d \end{bmatrix} \quad (4)$$

where $(u_x, u_y, u_z) = \vec{u}$ and *d* is (1 – cos θ). The matrix *R* is matrix for of the *Rodriguez rotation formula*.[4] It can be used to rotate a point, *v*, such that the rotated point, *v′*, is

$$v' = Rv. \quad (5)$$

This is represented in figure 11, where θ is assumed to be positive.

[4]Mason, M. T. *Mechanics of Robotic Manipulation*, Massachusetts Institute of Technology: The MIT Press, Cambridge, MA; Chapter 3, figure 3.26, 2001; 46.

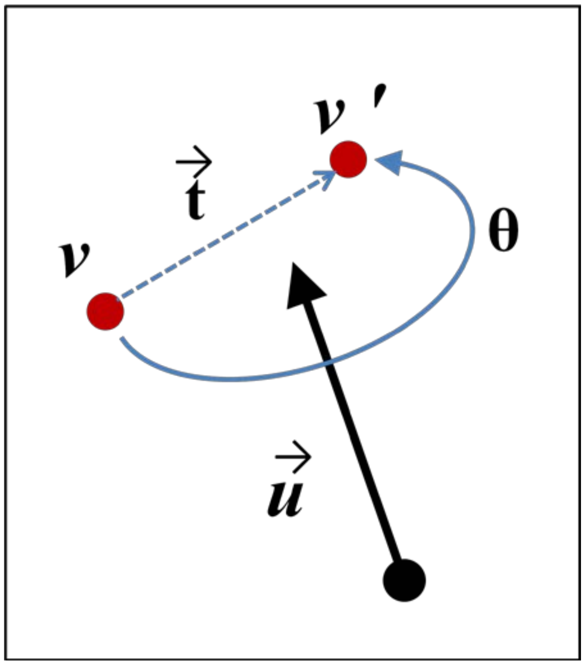

Figure 11. Simple rotation of a point about the origin.

Of course, the rotation of a point can also be expressed as a translation, call it *t*. Thus, if

$$t = Rv - v, \tag{6}$$

then v' can be defined as

$$v' = v + t. \tag{7}$$

5.2 Rotate a Point About Another Point Given a Rotation Matrix

Assume that one wanted to rotate about a point *p* instead of the origin. This requires two translations. The point *v* is translated first by *p*, effectively making *p* the origin. That point is rotated according to rotation matrix *R*. Finally, the rotated point is placed back into the original reference frame (a second translation of *p*). Thus *t* is defined as

$$t = R(v - p) + p - v. \tag{8}$$

This translation can then be applied to *v* as in equation 7.

Recall from section 3.1 Declarative Data, that the each joint is assigned a global translation vector. It begins with all zero entries. Every time *t* in equation 8 is calculated for a joint, we add it to its global translation vector. Thus, one can say that for *n* poses that affect joint *j*, then

$$T_j = \sum_{i=1}^{n} t_{i,j}, \tag{9}$$

where T_j is the global translation vector for joint *j*.

5.3 Rotate a Joint's Rotation Matrix

When a joint is rotated about another joint, its location may change. Regardless though, its rotation matrix will change. As discussed in section 3.1, number 2., each joint is assigned a rotation matrix. Assume that its rotation matrix is A. Then A is updated (symbolized by "→") such that

$$A \to RA \tag{10}$$

for a rotation matrix *R*, defined in section 4.1.

Similar to equation 9, after a sequence *n* poses (that affect joint *j*), the rotation matrix for joint *j* (represented as A_j) will be

$$A_j \to \prod_{i=1}^{n} R_{i,j}. \tag{11}$$

5.3.1 Posing Algorithm

We are ready to define a joint rotation (or posing) algorithm. First the following is defined:

- "joints"—the number of joints in the body
- "comps"—the number of components in the body
- "verts(c)"—the number of vertices in a component for a component with ID *c*
- "v(vx)"—the three-dimensional location of vertex *vx*
- "p(j)"—the three-dimensional location of joint *j*
- "$f(\vec{u}, \theta)$"—a function that returns the rotation matrix defined by $\vec{u}$ and θ; see equation 4.
- "$T(a)$"—a function that returns the global translation vector for joint with ID *a*
- "$A(a)$"—a function that returns the rotation matrix for joint with ID *a*
- "*"—matrix multiplication
- $C_{joint}(c)$ and $J_{tree}(i,j)$ as previously defined

The algorithm *Pose* is defined for the selection of a joint (*j*), an axis of rotation ($\vec{u}$), and an angle of rotation (θ) as (line numbers are provided in brackets and "//" indicates commented lines):

```
[ 1] Pose(j, u⃗, θ){
[ 2]   R = f(u⃗, θ) // Rotation matrix calculated
[ 3]   for (a = 1:joints){
[ 4]     if ($J_{tree}$(j,a) == 1){
[ 5]       // Translate affected joint
[ 6]       t = R*(p(a) – p(j)) + p(j) – p(a)
[ 7]       p(a) → p(a) + t
[ 8]       // Add t to T
[ 9]       T(a) → T(a) + t
[10]       // Rotate coordinate system of affected joint
[11]       A(a) → R*A(a)
[12]       // Check if a subcomponent is attached to the joint
[13]       // If so, rotate the subcomponent
[14]       for (c = 1:comps){
[15]         if ($C_{joint}$(c)== a){
[16]           for (vx = 1:verts(c){
[17]             t = R(v(vx) – p(j)) + p(j) – v(vx)
[18]             v(vx) → v(vx) + t
[19]           }
[20]         }
[21]       }
[22]     }
[23]   }
[24] }
```

The algorithm first calculates the rotation matrix using the axis and angle of rotation. For each joint in the body (line 3), if the joint is a child of joint *j* (line 4), then the child joint is relocated (line 7). The child joint's rotation matrix and global translation vector are also updated (lines 9 and 11). Finally, whatever component affected (line 15), all of its vertices need to be translated (line 18).

Note that vertices are changed for visualization purposes only. The Vertex Matrix file is never changed only its representation within the physical memory of the program.

5.3.2 Transforming a Body Into a Posture

A posture is defined as a terminating sequence of poses. However, now a more formal definition can be presented. Notice that the posing algorithm updates the global translation vector and coordinate system for certain joints (lines 9 and 11 of the *Pose* algorithm). Therefore, a terminating sequence of poses will produce final values for the global translation vector and rotation matrix for each joint. Thus, a posture is defined as a set of Translation (**T**) and Rotation (**A**) matrices.

$$P = \{\boldsymbol{T}, \boldsymbol{A}\} \tag{12}$$

This set represents a relatively small set of data needed to posture an object compared to recording a translation vector for each vertex. This is exactly how the posture is defined in the FragFly model.

Similar to the *Pose* algorithm, a *Posture* algorithm will be presented. The purpose of this algorithm is to translate the vertices of a body to match the predetermined posture found in the posture files. There is no way to load a posture into the GUI. However, this would be a very simple update to the GUI. The *Posture* algorithm will only modify the locations of the vertices in the subcomponents. Assuming one has access to the posture *P* as defined in section 5.3.1. Refer to the *Pose* pseudo-code for function definitions.

```
Posture($\mathbf{T}$,$\mathbf{A}$){
  for (s = 1:subcomponents){
    j = $C_{joint}$(s)
    R = $\mathbf{A}$(j)
    p = v(j)
    for (vx = 1:verts(s){
      t = R*(vert(vx) – p) + p + $\mathbf{T}$(j)
      vert(vx) $\rightarrow$ vert(vx) + t
      }
    }
  }
}
```

5.3.3 Reverse Posture Algorithm

Assume that one wanted to reverse the process of transforming a body into a posture. In the Posture algorithm, only the vertices of the appropriate component were translated. The reverse posture algorithm will add a layer of abstraction by returning a translation vector based on the inputs. The algorithm is defined this way so that the FragFly model can have a method to map the locations of the entry and exit points to a component to the ORCA reference frame.

The reverse posture algorithm, call it "*Unposture*," has three input arguments:

1. *v* —three-dimensional point
2. *c* —a subcomponent ID that *p* is associated with
3. *P* —the posture set

The *Unposture* algorithm returns one value *t*, which is a translation vector. This represents the translation of the point *v* to move it to the original orientation of the body part with ID *c*.

The following functions and variables will be used in the pseudo-code:

- "*P*(*T*,*j*)"—a function that returns the global translation vector for the joint with ID *j*
- "*P*(*A*,*j*)"—a function that returns the coordinate system for the joint with ID *j*
- "$(A)^T$ "—a unary operator that transposes the matrix *A*
- "*joint*(*j*)"—a function that provides the location of the joint with ID *j*

The algorithm *Unposture* thus defined as:

```
t = Unpose(v, c, P){
  j = $J_S(s)$
  t = $(P(A,j))^T * (v - (joint(j) + P(T,j))) + joint(j) - v$
  return t
```

The justification of this algorithm is left to the reader.

6. Conclusions

The FragFly model requires that users provide a number of input files associated with the human target. These files are part of a multibody system that was created that allows a body constructed of moveable parts to be manipulated. A GUI was created that simplifies the task of manipulating the body and provides real-time feedback. The details of the operations within the GUI were discussed, as well as the role the output files for the GUI play in the FragFly model.

The following are suggestions for future development:

1. Cosmetic changes to the GUI to promote ease of use.
2. Allow users to load postures into the GUI from posture files.
3. Include an "undo" button to take back the last set of changes for a body part.

Appendix A. “pose_body.m”

This appendix appears in its original form, without editorial change.

```matlab
function pose_body(v_file, f_file, joints_file, j2j_file, j2c_file)
    % Load data
    v = dlmread(v_file,',');
    f = dlmread(f_file,',');
    p2p =  dlmread(j2j_file,',');
    p2c = dlmread(j2c_file,',');
    leafj = p2c(end,:);
    p2c(end,:) = [];
    joints = dlmread(joints_file,',');
    org_j = joints;
    jR =   repmat([1 0 0 0 1 0 0 0 1],size(p2p,1),1);
    axor = repmat([1 0 0 0 1 0 0 0 1],size(p2p,1),1);
    thetas = zeros(14,3);
    cameratoolbar;
    %hax = axes('Parent',hfig);
    j = 1; ax = 1; theta = 0;
    %%%%%%%%%%%%%%%%%% Slider
    angle_text = uicontrol('Style','text',...
        'String','Angle: 0', 'Position',[20 60 100 20]);
    slider = uicontrol('Style', 'slider',...
    'Min',-180,'Max',180,'Value',0,...
    'Position', [20 40 300 20],'SliderStep',[1/360 10/360],...
    'Callback', {@slider_callback,angle_text});
    uicontrol('Style','text','String',['-150      -100      -50        0',...
        '        50        100     150  '], 'Position',[20 0 300 20]);
    uicontrol('Style','text','String',['           |            |       ',...
        '    |            |            |           |          |          '],...
        'Position',[20 20 300 20]);
    set(slider,'UserData',{joints,axor,jR,leafj,p2p,j,ax,org_j,v,f,...
        theta,p2c,thetas});
    graph_stuff(slider);
    legend({'Target','Joints','Selected Joint','Pitch','Roll','Yaw'},...
        'Location','NorthOutside')
        daspect([1 1 1]);
    %%%%%%%%%%%%%%%%%% Joint list
    j_text = uicontrol('Style','text',...
        'String','Joint:', 'Position',[20 160 40 20]);
    jlist = uicontrol('Style', 'popup',...
               'String', '1|2|3|4|5|6|7|8|9|10|11|12|13|14',...
               'Position', [20 110 40 50],...
               'Callback', {@jlist_callback, slider});
    %%%%%%%%%%%%%%%%%% Axor
    ax_text = uicontrol('Style','text',...
        'String','AXOR:', 'Position',[20 110 40 20]);
    axlist = uicontrol('Style', 'popup',...
               'String', 'Pitch|Roll|Yaw',...
               'Position', [20 60 80 50],...
               'Callback', {@ax_callback,slider});
    %%%%%%%%%%%%%%%%%% Commit Button
    comm_pb = uicontrol('Style','pushbutton',...
        'Position',[120 110 100 40], 'String', 'Commit','Callback',...
        {@commit_callback,slider});
    axis([-1 1 -1 1 0 2.2])
    set(gca,'view',[158.0000   13]);
end
%%%%%%%%%%%%%%%%%%% SUBROUTINES %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
    function slider_callback(hObj, eventdata, anglhand)
        ud = get(hObj,'UserData');
        joints = cell2mat(ud(1));
        axor =   cell2mat(ud(2));
        jR =     cell2mat(ud(3));
        leafj =  cell2mat(ud(4));
        p2p=     cell2mat(ud(5));
        j =      cell2mat(ud(6));
        ax =     cell2mat(ud(7));
        org_j =  cell2mat(ud(8));
        v=      cell2mat(ud(9));
        f =     cell2mat(ud(10));
        thetas = cell2mat(ud(13));
    % set(hObj,'Value',thetas(j));
    angle = round(get(hObj,'Value'))-thetas(j,ax);
    thetas(j,ax) = round(get(hObj,'Value'));
    p2c =   cell2mat(ud(12));
    theta = thetas(j,ax);
    [joints, axor, jR, R]=TRT(j,ax,angle,joints,axor,p2p,jR);
    trans = joints;
    % Apply each translation and rotation to each component
    for vr = 1:size(v,1)
        if (p2c(j,v(vr,4)) == 1)
            v(vr,1:3) = (R*(v(vr,1:3) - trans(j,:))' + trans(j,:)')';
        end
    end
    set(hObj,'UserData',{joints,axor,jR,leafj,p2p,j,ax,org_j,v,f,0,...
        p2c,thetas});
    set(anglhand,'String',['Angle: ',...
        num2str(round(get(hObj,'Value')))]);
    graph_stuff(hObj);
end
function jlist_callback(hObj, eventdata, shand)
    j= get(hObj,'Value');
    ud = get(shand,'UserData');
    ud(6) = num2cell(j); thetas = cell2mat(ud(13));ax = ud{7};
    cla; hold on;
    joints = cell2mat(ud(1));v=cell2mat(ud(9)); f =  cell2mat(ud(10));
    set(shand, 'UserData', ud);
    set(shand, 'Value',thetas(j,ax));
    graph_stuff(shand);
end
function ax_callback(hObj, eventdata, shand)
    x= get(hObj,'Value');
    ud = get(shand,'UserData');
    ud(7)=num2cell(x); j=ud{6};
    set(shand, 'UserData', ud);
    thetas = cell2mat(ud(13));
    set(shand, 'Value',thetas(j,x));
    graph_stuff(shand);
end
function commit_callback(hObj,eventdata,shand)
    ud = get(shand,'UserData');
    jR = cell2mat(ud(3));
    % C++ conversion
    for j = 1:size(jR,1)
        jR(j,:) = reshape(reshape(jR(j,:),3,3)',1,9);
    end
    trans = cell2mat(ud(1))-cell2mat(ud(8));
    v = cell2mat(ud(9));
    trans(:,3) = trans(:,3) - min(v(:,3));
    rot_file = uiputfile('rotations.txt','Save Rotation Matrices');
    if (rot_file ~= 0)
        dlmwrite(rot_file,jR,'delimiter',',','newline','pc');
    end
    trans_file = uiputfile('translations.txt','Save Translation Matrix');
    if (trans_file ~= 0)
        dlmwrite(trans_file,trans,'delimiter',',','newline','pc');
    end
    try
        % Using the rotate_human tool
        sv = dlmread('verts.txt',',');
        sf = dlmread('faces.txt',',');
        jR = dlmread(rot_file,',');
        trans = dlmread(trans_file,',');
        joints = dlmread('joints.txt',',');
        leafj = [1,13,14,2,3,5,4,6,8,10,7,9,12,11,13,1];
        for svr = 1:size(sv,1)
            lfc = leafj(sv(svr,4));
            R = reshape(jR(lfc,:),3,3)';
            sv(svr,1:3) = ...
            (R*(sv(svr,1:3)-joints(lfc,:))'+joints(lfc,:)')'+trans(lfc,:);
        end
        cla; hold on;
        patch('vertices',sv(:,1:3),'faces',sf(:,1:3)+1,'facealpha',1,...
            'edgealpha',0.005,'facecolor','g','facelighting','phong');
        daspect([1 1 1])
    catch
        errordlg('Could not find files or you cancelled while creating files.');
    end
end
function graph_stuff(slider)
    cla; hold on;
    ud = get(slider,'UserData');
    % Graph patches
    v=cell2mat(ud(9)); f=cell2mat(ud(10));
    patch('vertices',v(:,1:3),'faces',f(:,1:3)+1,'facealpha',.05,...
        'edgealpha',0.05,'facecolor','g','facelighting','phong');

    % Graph joints
    joints = cell2mat(ud(1));
    scatter3(joints(:,1),joints(:,2),joints(:,3),'bo','filled');
    j = ud{6};
    scatter3(joints(j,1),joints(j,2),joints(j,3),'ro','filled');
    % Add text labels
    ofst = .04;
    for jj=1:size(joints,1)
        text(joints(jj,1)+ofst,joints(jj,2)+ofst,joints(jj,3)+ofst,...
            num2str(jj),'FontSize',10);
    end
    jR = cell2mat(ud(3));
    scl = .4;
    %if (ud{7} ==1)
    plot3([joints(j,1)   joints(j,1)+jR(j,1)*scl],...
          [joints(j,2)   joints(j,2)+jR(j,2)*scl],...
          [joints(j,3)   joints(j,3)+jR(j,3)*scl],'r-',...
          'Linewidth',1+2*(ud{7}==1));
    %elseif (ud{7} == 2)
      plot3([joints(j,1) joints(j,1)+jR(j,4)*scl],...
          [joints(j,2)   joints(j,2)+jR(j,5)*scl],...
          [joints(j,3)   joints(j,3)+jR(j,6)*scl],'b-',...
          'Linewidth',1+2*(ud{7}==2));
    %else
      plot3([joints(j,1) joints(j,1)+jR(j,7)*scl],...
          [joints(j,2)   joints(j,2)+jR(j,8)*scl],...
          [joints(j,3)   joints(j,3)+jR(j,9)*scl],'k-',...
          'Linewidth',1+2*(ud{7}==3));
    %end
    joints = find(ud{1,5}(ud{1,6},:));
    for j=1:size(joints,2)
        comps = find(ud{1,4}==joints(j));
        for c = 1:size(comps,2)
            patch('vertices',v(:,1:3),'faces',f(f(:,4)+1==comps(c),1:3)+1,'facealpha',.5,...
                'edgealpha',0.1,'facecolor','g','facelighting','phong');
        end
    end
    light
    daspect([1 1 1])
end
```

Appendix B. “TRT.m”

This appendix appears in its original form, without editorial change.

```
function [pivot, axor, jR, R] = TRT(k, x, rads, pivot, axor, p2p, jR)
% Find the rotation matrix R, and translate, rotate, and translate affected joints
R = zeros(3);
rads=rads*(pi/180);
c=cos(rads);d=1-c;s=sin(rads);
v0=axor(k,3*(x-1)+1);
v1=axor(k,3*(x-1)+2);
v2=axor(k,3*x);
  R(1)=v0*v0*d+   c ; R(4)=v0*v1*d-v2*s ; R(7)=v0*v2*d+v1*s ;
  R(2)=v1*v0*d+v2*s ; R(5)=v1*v1*d+   c ; R(8)=v1*v2*d-v0*s ;
  R(3)=v2*v0*d-v1*s ; R(6)=v2*v1*d+v0*s ; R(9)=v2*v2*d+   c ;
for p = 1:size(p2p,2)
    if (p2p(k,p) == 1)
        % Transform affected pivot points
        pivot(p,1:3) = (R*(pivot(p,1:3) - pivot(k,:))' + pivot(k,:)')';
        % Transform affected axes
        for jj = 1:3
            tax = R*(axor(p,(3*(jj-1)+1):3*jj))';
            axor(p,(3*(jj-1)+1):3*jj) = tax/norm(tax);
        end
        % The jR matrices are stored row-major. So we need to transpose
        % them in order to multiply it by R
        jR(p,:) = reshape((R*reshape(jR(p,:),3,3)),1,9);
    end
end
```

NO. OF COPIES	ORGANIZATION
1 (PDF)	DEFENSE TECHNICAL INFORMATION CTR DTIC OCA
1 (PDF)	DIRECTOR US ARMY RESEARCH LAB IMAL HRA
1 (PDF)	DIRECTOR US ARMY RESEARCH LAB RDRL CIO LL
1 (PDF)	GOVT PRINTG OFC A MALHOTRA
1 (PDF)	RDRL WML A B FLANDERS

INTENTIONALLY LEFT BLANK.